NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

A DESIGN FOR SENSING THE BOOT TYPE OF A TRUSTED PLATFORM MODULE ENABLED COMPUTER

by

Richard C. Vernon

September 2005

Thesis Advisor:	Cynthia E. Irvine
Second Reader:	Timothy E. Levin

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** September 2005	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE: A Design for Sensing the Boot Type of a Trusted Platform Module Enabled Computer			**5. FUNDING NUMBERS**
6. AUTHOR(S) Richard C. Vernon			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited			**12b. DISTRIBUTION CODE**

13. ABSTRACT (maximum 200 words)

Modern network technologies were not designed for high assurance applications. As the DOD moves towards implementing the Global Information Grid (GIG), hardened networks architectures will be required. The Monterey Security Architecture (MYSEA) is one such project.

This work addresses the issue of object reuse as it pertains to volatile memory spaces in untrusted MYSEA clients. When a MYSEA client changes confidentiality levels, it is possible that classified material remains in volatile system memory. If the system is not power cycled before the next the login, an attacker could retrieve sensitive information from the previous session. This thesis presents a conceptual design to protect against such an attack.

A processor may undergo a hard or soft reboot. The proposed design uses a secure coprocessor to sense the reboot type of the host platform. In addition, a count is kept of the number of hard reboots the host platform has undergone. Using services provided by the secure coprocessor, the host platform can trustfully attest to a remote entity that it has undergone a hard reboot. This addresses the MYSEA object reuse problem. The design was tested using the CPU simulator software SimpleScalar.

14. SUBJECT TERMS Information Assurance, Monterey Security Architecture, Object Reuse, Trusted Platform Module			**15. NUMBER OF PAGES** 68
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

A DESIGN FOR SENSING THE BOOT TYPE OF A TRUSTED PLATFORM MODULE ENABLED COMPUTER

Richard C. Vernon
Civilian, Naval Postgraduate School
B.S., University of Arkansas - Fayetteville, 2003

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
Month Year

Author: Richard C. Vernon

Approved by: Cynthia E. Irvine, Ph.D.
Thesis Advisor

Timothy E. Levin
Second Reader/Co-Advisor

Peter J. Denning, Ph.D.
Chairman, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Modern network technologies were not designed for high assurance applications. As the DOD moves towards implementing the Global Information Grid (GIG), hardened networks architectures will be required. The Monterey Security Architecture (MYSEA) is one such project.

This work addresses the issue of object reuse as it pertains to volatile memory spaces in untrusted MYSEA clients. When a MYSEA client changes confidentiality levels, it is possible that classified material remains in volatile system memory. If the system is not power cycled before the next the login, an attacker could retrieve sensitive information from the previous session. This thesis presents a conceptual design to protect against such an attack.

A processor may undergo a hard or soft reboot. The proposed design uses a secure coprocessor to sense the reboot type of the host platform. In addition, a count is kept of the number of hard reboots the host platform has undergone. Using services provided by the secure coprocessor, the host platform can trustfully attest to a remote entity that it has undergone a hard reboot. This addresses the MYSEA object reuse problem. The design was tested using the CPU simulator software SimpleScalar.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would like to thank my advisors Dr. Irvine and Prof. Levin for their contributions of time, patience, and expertise without which this thesis would be possible. I would further like to thank Dr. Irvine and the National Science Foundation for providing the CyberCorp Scholarship under which I which I have studied at the Naval Postgraduate School.

This material is based upon work supported by the National Science Foundation under Grant No. DUE-0114018 and Grant No. CNS-0430566. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

Many modern network architectures were created with inadequate regard to security concerns. Currently, many of these technologies have been deployed in situations with high information assurance requirements. As the DOD moves to implement its Global Information Grid (GIG) initiative, new networked security architectures will need to be developed. One such project is the Monterey Security Architecture (MYSEA).[1] The MYSEA project leverages existing COTS products to implement multilevel networks over which secure information services can be provided.

This project addresses a specific security issue within the MYSEA framework. If a client computer has been rebooted without being power cycled, it is believed that sensitive information may remain within volatile memory components of the system. This includes main system RAM. This information is then vulnerable to retrieval by malicious programs or users.

To address this issue, this thesis hypothesizes a design that enhances a client workstation so that it senses the type of reboot it underwent. This design is based on the Trusted Computing Group's Trusted Platform Module.[2] The Trusted Platform Module (TPM) is a secure coprocessor providing security and encryption services to the host platform. A modified TPM is used to sense the type of reboot a computer has undergone. This functionality will be shown to be correct using the validation properties of a Reference Monitor, which are that is understandable, cannot be bypassed, and cannot be modified.[3] Furthermore, the design allows the TPM to keep a count, the Boot Odometer Value, which tracks the number hard reboots the host platform has undergone. Used in accordance with remote attestation, the computer can then trustfully prove to a remote entity the type of reboot it has undergone. In addition, this thesis presents a proof of concept simulation of the design using the CPU hardware simulation software SimpleScalar.

This thesis first provides background information, in the second chapter, on the MYSEA project, remote attestation, sealed storage, and hardware emulation. The third chapter presents the design of the system used to sense the boot type of the host platform.

Software requirements for utilizing the new design mechanism are also included. At the end of the third chapter is a short section on possible applications of the design. The fourth chapter describes the development and testing of the simulation used to demonstrate certain properties of the design. The fifth chapter provides analysis of implications associated with hypothesized design. The sixth chapter concludes this document with a discussion of further work and summary.

II. BACKGROUND

The goal of this thesis is to address the object reuse problem of MYSEA and similar architectures by using remote attestation based on a new hardware primitive, and to develop a proof of concept of such a product using hardware simulation. Background concepts supporting this goal are discussed in this chapter.

A. MYSEA PROJECT OVERVIEW

The MYSEA (Monterey Security Architecture) Project is a trusted, network-based architecture designed for securely sharing information in dynamic, multi-domain environments. The MYSEA framework provides a full featured multilevel secure LAN that can interact with single security level legacy networks (NIPRNET, SIPRNET, etc.). It is based on commercial off-the-shelf (COTS) components allowing the DOD to leverage existing hardware and software, helping to preserve its investment in such products. The MYSEA Project relies on high assurance design and development methods to ensure the integrity and trust of the system.[4]

At the core of the MYSEA framework are the MYSEA Server and the Trusted Path Extension (TPE). The MYSEA server operates as both a multilevel security (MLS) policy enforcement mechanism and platform on which information services can be delivered and consumed via standard web-enabled tools. The MYSEA design allows confederated servers to distribute the load of multiple network services. The base MYSEA server is currently implemented on a DigitalNet XTS-400. The XTS-400 is a high assurance hardware/OS platform that enforces an MLS security policy through the use of labeled subjects and objects. The TPE is a handheld computing device running a lightweight security kernel. It provides an unspoofable link from the user to the MYSEA Server over which a variety of security services can be delivered. In addition, the TPE mediates access from the users workstation to the server.[5] Figure 1 displays the working model for the MYSEA. The trusted channel modules provide label integrity for entire single security level legacy networks. They provide the network an unspoofable link to the MYSEA server over which authentication and other security services can be delivered. Furthermore, Figure 1 shows high assurance link encryptors than can be integrated if required.[1]

1. Trusted Path Extension

The Trusted Path Extension (TPE) is the basis for providing security services from the MYSEA Server to the client user. Through this link the MYSEA Server provides a variety of security critical operations. These include a secure attestation key, trusted path services, controlled LAN access, communications and cryptographic services, negotiated session services, and control of security critical activities. The TPE is based on a high assurance separation kernel running on a dual NIC PDA. The TPE protects itself from malicious software operating on the client computer. This is done via the physically separated execution environment and the high assurance kernel that prevents network vector exploits. Furthermore, the MYSEA architecture does not depend on the trustworthy behavior by the client machine. All trust is derived from the relationship between the TPE and the MYSEA Server.[4]

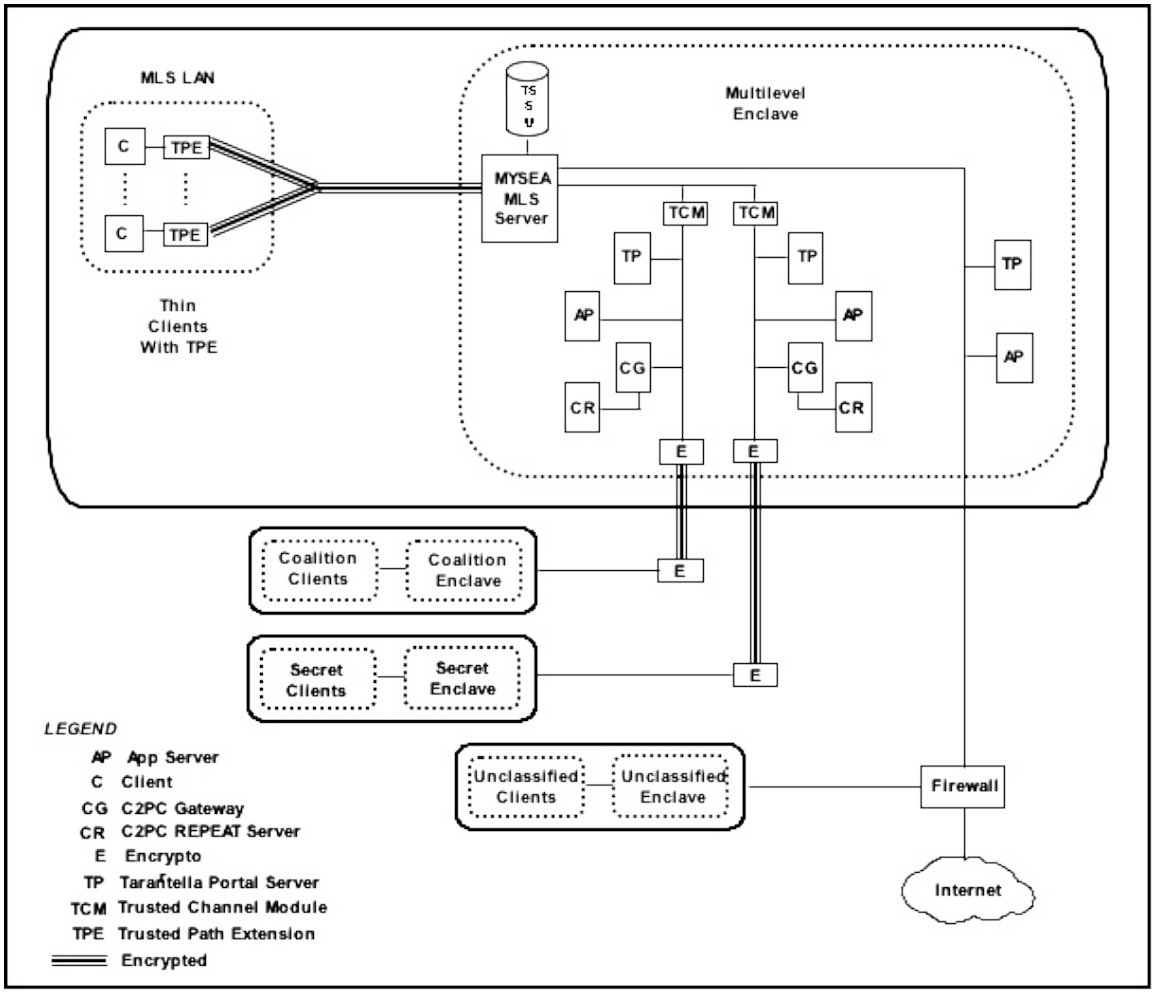

Figure 1. MYCEA Architectural Overview taken from [1]

2. Object Re-use Considerations

Object reuse must be addressed in secure systems to prevent unintentional access to information by processes, and ultimately users, with different security attributes. Object reuse ensures that when storage resources are allocated to a process, they have been purged of any residual information that may be remnant from a previous process.[6] In multi-user systems that implement Discretionary Access Control (DAC) policies, mechanisms that address object reuse prevent information from being leaked between processes that do not wish to share information. Unfortunately, many modern operating systems with DAC policies fail to fully address object reuse. Multilevel Security (MLS) systems operating under Mandatory Access Control (MAC) policies must implement strict mechanisms for ensuring correct object reuse. Because both high and low sensitivity objects exist within the same system, the possibility exists that a low process may be able read residual high information if an allocated resource is not properly purged.

For example, suppose a process operating at a high security level in an MLS system "deleted[1]" information from a hard drive and released the space back to the operating system. Now suppose a second process running at a lower security level requests space on the hard drive from the operating system and receives space formerly allocated to the high level process. If the high level information wasn't properly removed from the hard drive, the lower level process might be able to retrieve the higher level information retained on the hard drive.

The main RAM and other volatile memory locations on the main circuit board of MYSEA clients present possible vectors through which object reuse weaknesses could be exploited. The MYSEA client OS is not trusted to separate information at different security levels. Because the MYSEA client hardware does not clear RAM after a soft reboot, sensitive information could possibly be retained in memory. A malicious user could exploit this situation if only a soft reboot was permitted. The user could log in at a

[1] In this case, we assume deletion doesn't destroy a file. It merely marks the space occupied by the file as available for reallocation by the OS.

lower classification level and then use administrative tools to access RAM as a raw device. It would then be possible to reconstruct sensitive information left over from a previous higher sensitivity session.

B. REMOTE ATTESTATION

Most modern, networked software applications implicitly trust the integrity of other systems they communicate with over a network. Even if the software application is trusted, there is no way to verify the integrity of the operating environment of a remote host. When queried for such state information, a malicious host can lie with impunity. For example, in peer-to-peer (P2P) file trading networks, copyright holders frequently flood the network with garbage files that have metadata that matches more valuable files. The distribution of such files is achieved through the use of non-standard clients that act maliciously on the network. Unfortunately, legitimate clients cannot differentiate between trusted and malicious peers. Users waste resources downloading garbage files. This reduces the overall utility of the network.

Secure coprocessors provide methods for a computer to trustfully attest to a remote host. Trusted remote attestation is the process by which a host can provide high integrity evidence of its current operating environment. This can include the status of hardware, operating systems, and running software. Using this information, remote hosts can make decisions as to whether or not to provide services and data to the attesting host. In the case of a P2P networks, non-standard clients could be effectively disallowed through the use of trusted attestation. Every time a host wished to join the network, a peer would force the joining host to attest to its current operating state. Only systems with the ability to reliably attest would be able to join the network. Non-standard and malicious clients would be excluded, to the extent that the integrity of the secure coprocessors and their integrity infrastructure remains intact.[7]

1. Attestation and Trusted Computing

The Trusted Computing Group's Trusted Platform Module (TCG TPM) is a secure coprocessor that provides essential security services such as key storage and fast hardware encryption to the host devices in which it is embedded.[8] One such service is trusted remote attestation. To provide trusted remote attestation to the host platform, the TPM uses a combination of system integrity measurements and public key cryptography.

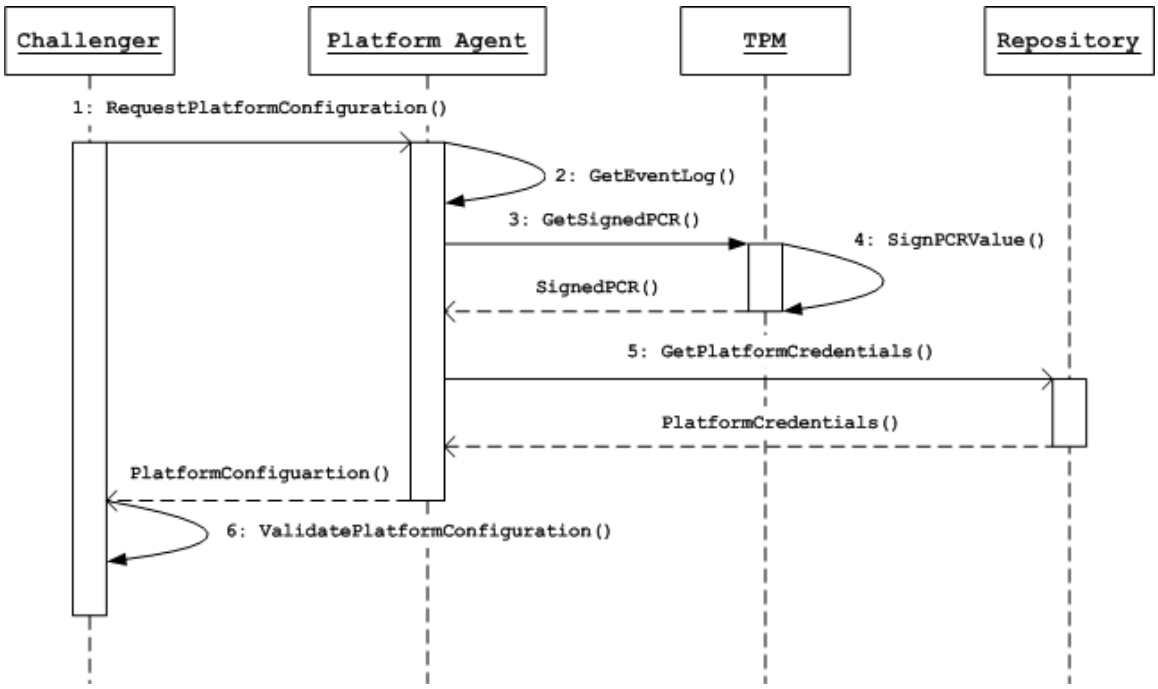

Figure 2. Generic description of the TPM Remote Attestation process taken from .[2]

Before the attestation process begins, a client must first make a request to a server for information or services. If the server wishes to validate the operating environment of the client, it challenges the client to perform a TPM-based trusted attestation. Because the complete attestation object is made up of several pieces of information, multiple steps are required to complete the process.

In Figure 2, the server is referred to as the Challenger. It begins the attestation process by requesting an attestation from the client, which is labeled Platform Agent in Figure 2. This is shown with the command *RequestPlatformConfiguration()*. The client, in response to the challenge being sent, begins a set of operations that generate the attestation response.

In the second step in Figure 2, the client retrieves information from the Stored Measurement Log (SML) using the command *GetEventLog()*. The SML is an event log that records sequences of references to measurement values and measurement digests. Measurement values are representations of system data or stored programs. Measurement digests are cryptographic hashes of the measured values. Measuring a value is equivalent to generating a hash of the value. Sequences of measurements can be generated using

measurement extension. The current digest is concatenated with the value to be measured. The concatenated object is then hashed and stored. This allows arbitrarily long measurement sequences to be stored in a limited amount of space.[9] By storing references to the measured objects and the storage locations, the SML can generate a digest at any point in the measurement sequence.

The third step shows the Platform Agent performing the request *GetSignedPCR()*. A Platform Control Register (PCR) is a volatile register inside the TPM where it stores measurement digests generated with the SHA-1 algorithm. The first eight PCR registers are used to store the measurements of system initialization components such as the BIOS, expansion ROM's, and boot sector.[10] The other PCR's are used to store other measurements of system code and firmware as required. These measurements are referred to as the *state* of the system. The PCR values represent the most currently measured state of the host platform. By retrieving a signed PCR from the TPM, the Platform Agent can trustfully attest to the state of the host platform. At any given time, the SML can be used to verify the value currently occupying a PCR.

In the fourth step, the TPM performs the function *SignPCRValue()*. This is the first place where trust is derived in the attestation process. The TPM uses keys that are protected from the host platform to sign its PCR. Key signing is performed on a processor embedded in the TPM that is separate from the host platform's CPU. By protecting the signing keys from the host platform, the Challenger can assume that any measurements signed by the TPM are valid. Without access to the signing keys, there is no way for the host platform to maliciously lie to the Challenger about its operating state.

The fifth step allows the TPM to request that a third party vouch for the authenticity of its public key. This validation is required because the TPM may use many such keys in the process of attesting to different remote entities. The third party is labeled the Repository in Figure 2. Each TPM is loaded at the factory with a key pair that is unique to it. The Repository encrypts information with the TPM's public key such that only an individual TPM can decrypt it with its corresponding private key. In this way, the repository can sign a TPM's public attestation keys and return them encrypted so that only that TPM can decrypt them. When a TPM uses those attestation keys, it can pass

along the signed public keys vouched for by the Repository. The Challenger can then trust the public keys used to authenticate the signed attestation information.

The final step is for the Challenger to verify all the information provided to it by the Platform Agent.

2. Attestation Research

The attestation procedure above has a privacy flaw whereby the Challenger can obtain the unique identify of the TPM that attests to it. The Challenger must collude with the Repository to accomplish this. When the Challenger receives the attestation package from the Platform Agent, it simply has to ask the colluding Repository to provide it with the identity of the TPM that previously submitted the attestation key for verification.

Direct Anonymous Attestation (DAA) can be used to combat the issue of collusion. Described in a paper by Brickell, et al, DAA is a complicated protocol that utilizes resource intensive cryptographic procedures. It is currently being integrated into the newest version of the TPM specification.[11]

C. SEALED STORAGE

In general, if data must be stored on a hard drive with high integrity and high secrecy, the data might be encrypted with a key known only to authorized entities. However, in a general-purpose computer, the program has no way of truly knowing if malicious programs are sniffing input from the user. If this were the case, even encryption would not protect the data because the malicious program would have the encryption key. However, using the TPM as a way to encrypt data and store keys that are protected from untrusted parts of the system, data can be securely stored on the hard drive.

Sealed Storage is the tool that programs will use to securely store large information blocks that need to be protected from malicious software that may be running on the host platform. While the TPM interface is large and complicated, it is generally expected that applications will access TPM functionality through a TPM library that presents high-level function calls. In this way, the application developer will be shielded from many of the subtle TPM intricacies. Simply put, Sealed Storage works by depending on the TPM to accurately measure the state of the machine. From the TPM's

perspective, the state of the machine is based on measurements of system components and stored software applications. The measurements are stored in PCR registers. The collection of values in the PCR registers make up the TPM's concept of host platform state. The TPM generates a session key based on the state of the machine. Only if the machine matches the state of the machine when the data was encrypted will it generate the correct decryption key. This means an attacker can't remove the hard drive and use a machine he controls to extract secret data from the drive.

D. HARDWARE SIMULATION

Hardware simulation is the ability to represent the operation of a discrete hardware device in software. The software-based simulator keeps track of the operating states of the target hardware as it executes. Studying the empirical data generated from running the simulator can validate or disprove hardware design assumptions.

Hardware simulation provides a useful, flexible development environment for both software and hardware designers. Software designers are able to do meaningful development and testing for platforms that may not yet exist as hardware. Hardware designers have the ability to test and tweak their designs before going through a costly fabrication process. While a plethora of simulators may exist for a given hardware platform, each will have relative merits depending on the requirements of the user.

The three main characteristics of a hardware simulator are performance, flexibility and detail.[12] Maximizing one characteristic requires a tradeoff in the utility of another. Because host hardware may have little in common with the hardware being simulated, the performance characteristic is often the first addressed. By focusing on performance, simulator developers can decrease the required computing resources for useful development. For those developing new hardware designs, the ease with which a simulator can be changed to incorporate new designs or instructions is often the key characteristic of interest, but this may affect the performance or level of detail the simulator is able to achieve. For those doing computation research with simulators, the granularity, or detail characteristic, of the simulation is often of great importance. However, the more that the simulator reproduces the entire design and keeps track of machine state variables as it executes, the more performance and flexibility may suffer.

There are two distinctive paradigms for simulator construction – trace-driven or execution-driven. Trace driven simulators use program traces to drive a simulation engine. A trace is a recording of the complete operation of a program as it executed. All instructions, inputs, and outputs are recorded. Traces allow a hardware designer to recreate the same test environment for multiple hardware designs. Design changes can be closely scrutinized for performance issues.

An execution-driven simulator uses program binaries to drive the execution engine. Complete program behavior is not known *a priori*. Programs that require external input will produce varying execution statistics depending on the input.

1. SimpleScalar

SimpleScalar is an open-source, execution-driven CPU simulator. It is built on a highly flexible code base that makes it easy to implement changes to the underlying hardware descriptions of the target platforms. Support for several CPU instruction set architectures is currently available. These include Alpha, MIPS, PowerPC, and x86.

Included in the SimpleScalar suite are a variety of simulators covering a range of complexity.[13] There are simulators optimized for speed that provided serialized, in-order execution. There is a cache hierarchy simulator. The most complex simulator includes a micro-architecture with branch prediction and out of order execution. Each simulator focuses on a functional characteristic of either performance or detail. Each simulator collects a wide array of information and statistics about executed processes that detail the operating characteristics of the simulate hardware.

E. MYSEA & THE BOOT ODOMETER CONCEPT

The overall MYSEA architecture operates in such a way as to mitigate the risk caused by an untrusted client. While the TPE can provide security services directly between the user and the MYSEA Server, neither has any way of validating a hard reboot of the client computer. Without a hard reboot, there remains a potential object reuse that could provide an exploitable vulnerability to a determined adversary. This occurs when the client workstation is not rebooted to clear memory between a high confidentiality session and a low one. Currently, procedural policy requires that users hard reboot the system when switching between session levels. Because users may not always follow

expected procedures, it is possible that a malicious program could retrieve sensitive information from memory after a user failed to hard reboot the client machine.

To alleviate the vulnerability, this thesis proposes a method by which changes to a secure coprocessor, the TCG TPM, are used to detect the type of reboot the host (i.e. the client) platform underwent and to keep a count of the number of times the host platform has undergone a hard reboot. An instruction is added to the TPM that allows local programs to query for the current number of hard reboots that have been observed. These changes are augmented by remote attestation, which allows the MYSEA Server to query the host regarding the number of hard boots that the host has undertaken to date. These changes are made to the TPM firmware and have been evaluated using SimpleScalar.

There are several reasons why the attestation on the reboot status is done between the client and the MYSEA Server instead of between the client and the TPE. First, the TPE is a PDA style device with limited resources in the areas of processing and main memory. Attestation requires intensive computations for the public key cryptography system. This would further stress the TPE and take away resources needed to establish secure communications with the MYSEA server. Second, adding attestation-checking abilities to the TPE would increase its complexity. Added complexity reduces the assurance that the designers can prove the correctness of the design and operation of the TPE. Third, the TPE is not integrated into the client computer main board. Consequently, there is no way for the TPE to directly observe if the client has cycled through a hard reboot. Although integration of the TPE and the client computer has been considered, the system would demand a much more complex trusted computing base than the relatively simple separation kernel currently under development. [14]

This chapter provided a brief overview of the topics referred to in this thesis including attestation, hardware simulation, and the MYSEA. It also discussed the relevance of the main thesis topic to the MYSEA project. The next chapter will elaborate the conceptual design that addresses the object reuse problem. The next chapter also briefly explains several scenarios in which the conceptual design can be used.

III. CONCEPTUAL DESIGN

A. CONCEPT OF OPERATION

The Boot Odometer concept arises from the need to reliably attest to a remote entity that a local computer has been hard rebooted (i.e. power cycled) as opposed to soft rebooted (i.e. software initiated reboot). A hard reboot indicates to a remote entity that certain board level components such as memory have been cleared due to power loss.

To achieve this, a secure coprocessor is added to the main system board. A secure coprocessor provides a highly trustworthy execution environment for the storage of security-critical information and for security-critical computation. The secure coprocessor stores a high integrity number known as the Boot Odometer Value. This value keeps track of the number of times the computer has been hard rebooted. Every time the computer hard reboots, the secure coprocessor causes the Boot Odometer Value to increment by one. The secure coprocessor does not increment the Boot Odometer Value in the case of soft reboots.

A host equipped with a secure coprocessor performs a trusted remote attestation to prove to a remote entity that it has undergone a hard reboot. The host requests to its secure coprocessor that it sign the Boot Odometer Value using a private key known only to the secure coprocessor. The host then packages the signed value and sends it to the remote entity. The remote entity uses the corresponding public key to validate the Boot Odometer Value. In addition, the remote entity believes that the host's secure coprocessor protects its private signing keys from tampering or observation by the host. This allows the remote entity to trust with high assurance that the Boot Odometer Value is authentic because without the signing key, there is no way for the host to forge the signature. By comparing the current Boot Odometer Value to previously known values, the remote entity can confirm the hard reboot attested to by the host.

B. HIGH LEVEL SYSTEM REQUIREMENTS

The purpose of the Boot Odometer Value is to ensure the computer has been powered cycled. This entails several high level requirements for correct action. These are listed below.

- The Boot Odometer Value (BOV) must increment every time a full power cycle-boot sequence completes.
- The Boot Odometer Value must not increment during a software reboot or at any other time except for a hard reboot.
- The Boot Odometer Value must be protected when the system is in a power off state. This includes accidental and malicious tampering.
- The Boot Odometer Value must be protected from software tampering during normal host computer operation.
- The increment operation must be atomic. It should not be interruptible by power cycles initiated during the boot sequence.
- The secure coprocessor must be able to detect exceptions in the Boot Odometer mechanism.
- The secure coprocessor must be able to handle exceptions in the Boot Odometer mechanism.
- The secure coprocessor must be able to attest to the Boot Odometer Value.

Each of these requirements is discussed below.

The Boot Odometer Value must increment every time a power cycle-boot sequence occurs. The requirement fulfills the overall goal of being able to ensure that the computer was power cycled. No requirement is made regarding the value of the increment; however, an increment of one is preferred to minimize the number of rollover events.

The increment operation must be atomic. This is to prevent the machine from entering an ambiguous state. Since the operation of the Boot Odometer is security relevant, it must execute fully or not at all.

The Boot Odometer Value must be protected in non-volatile storage, since the value of the Boot Odometer must be retained without power. The storage must be resistant to both physical and software-based attacks.

The Boot Odometer Value must be protected from changes during normal host computer operations. The host computer that houses the secure coprocessor will use the coprocessor for a variety of operations. Access to the secure coprocessor is negotiated via software calls. It must not be possible to use any combination of software calls to the secure coprocessor to change the Boot Odometer Value.

The secure coprocessor must be able to detect exceptions in the Boot Odometer mechanism. Because the Boot Odometer mechanism is part of the system boot routine, exceptions in its operations can affect system security by corrupting the boot process. By detecting exceptions, the system can take appropriate action to ensure system confidentiality or integrity.

The secure coprocessor must be able to handle exceptions in the Boot Odometer mechanism. Exception handling must be well defined and not leave the computer in an insecure state.

The secure coprocessor must be able to attest the status of Boot Odometer Value. This is a requirement for operating in high assurance networks.

The Boot Odometer Value must not change during a software reboot. A software reboot does not ensure power loss to the machine.

The Boot Odometer operates in the following manner. If the platform has undergone a hard reboot, power to the TPM will have been interrupted causing the PCR registers inside the TPM to initialize to a known value. For simplicity, it is assumed that the initialization value is all zeros. If the platform has undergone a soft reboot, the contents of the PCR registers will not have been reset. During its initialization phase, the TPM inspects the PCR where it had previously placed the Boot Status Indicator. If the Indicator appears inside the specified PCR, rather than the initialization value, then the TPM assumes that it has undergone a soft reboot and the TPM is disabled. In the case where computer underwent a hard reboot, the TPM first updates the contents of the specified PCR with the Boot Status Indicator, then fetches the Boot Odometer Value, increments it, and finally places it back into the long-term storage of the TPM. The initial

value of the Boot Odometer Value is zero. This value is initialized during factory production of the TPM platform. The non-volatile memory and the PCR in which the Boot Status Indicator is stored are untouchable by the host platform CPU. This protects them against malicious alteration.

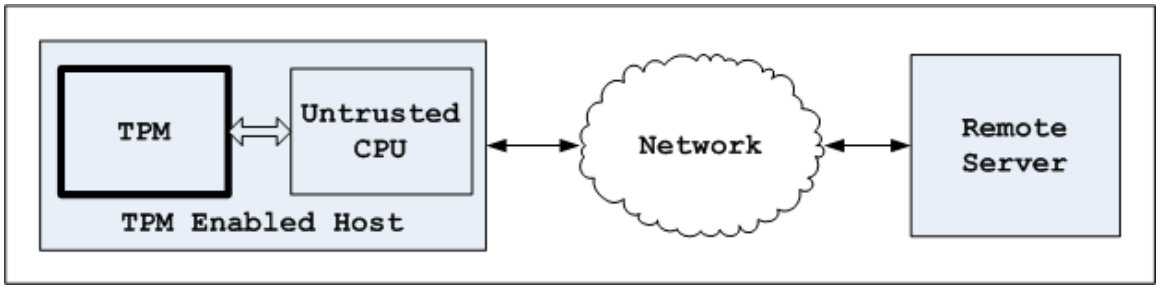

Figure 3. Simple Attestation Layout

A remote entity has two choices when requesting trusted attestation regarding the type of reboot a host underwent. First, the remote entity can store a previously known value of the Boot Odometer Value. It can then request the current Boot Odometer Value. If the current value is larger than the stored value, then the remote entity can assume that the host has undergone a hard reboot. Secondly, it can ask the host to directly attest to the fact that it has undergone a hard reboot more recently than a soft reboot. In this case, the TPM would inspect itself and if it were not disabled, then it would provide a positive attestation of the fact. In the design presented here, a soft reboot will disable the TPM. Thus, the fact the TPM is enabled is evidence that the most recent boot that platform has undergone is a hard reboot. Figure 3 shows a simple division between the TPM and the Host CPU. The Remote Server trusts the TPM to the degree that it has been evaluated to shield its cryptographic secrets and computations from the host platform. Using the underlying cryptographic services, the TPM can trustfully attest to its environment.

C. HARDWARE REQUIREMENTS

The Boot Odometer mechanism is implemented by making changes to the initialization firmware within the Trusted Computing Group's Trusted Platform Module (TCG TPM)[2]. The TPM is an open, industry-developed standard for the creation of secure co-processors in diverse computing platforms. A logical overview of the components inside the TPM can be seen in Figure 4.

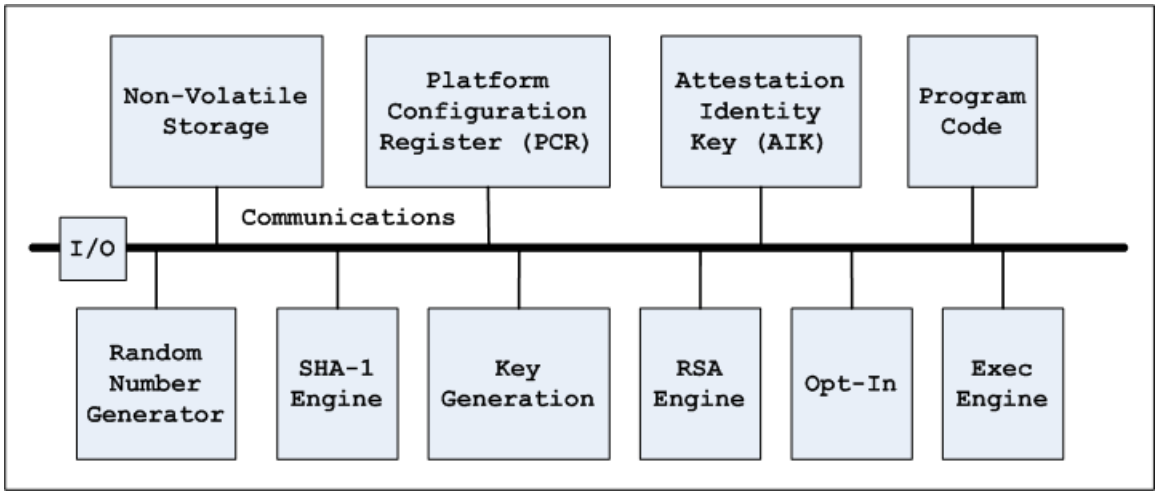

Figure 4. Logical component architecture of the TPM secure co-processor taken from [2]

Currently, many computer manufacturers are in the process of adding TPM's to their consumer and business model PC's. The changes required for the BOV will affect a target software function that exists within the TPM firmware. The name of the function is TPM_Startup. TPM_Startup executes every time the TPM goes through an initialization cycle. This makes it an ideal candidate for housing the Boot Odometer functionality.

The TPM_Startup function call is part of the TPM initialization phase.[15] At the beginning of every boot cycle, the TPM undergoes a transition function called TPM_Init. TPM_Init transitions the TPM into its first stage of initialization. This function behaves identically whether the system underwent a hard or soft reboot. TPM_Init places the TPM in a state where it waits for the command to execute TPM_Startup. Platform initialization code must inform the TPM what type of initialization it is currently undergoing. The TPM_Startup function behaves differently based on one of three flags that signal its intended operation. The TPM_ST_CLEAR flag signals the TPM to reset volatile TPM variables back to their default states. The TPM_ST_SAVE flag signals the TPM to restore volatile variables back to their previously known values. This occurs when the computer starts from a hibernation related state. The TPM_ST_DEACTIVATED flag signals the TPM to enter into a deactivated state.[15] Because we are only interested in the situation were the computer boots into fully operation mode from a power-off state, we only consider the case were TPM_Startup is called with the TPM_ST_CLEAR flag.

The TPM Specification requires that all system boots must first start with a system wide reset. This includes physically signaling system components that a system boot is happening. This requirement prevents the TPM from being maliciously reset without the rest of the platform also being reset. If this were to occur, the TPM would be in a state where it would be vulnerable to certain masquerade attacks.[15]

The required actions taken by the TPM when executing TPM_Startup with the TPM_ST_CLEAR flag are laid out in a standard format in the TPM specification document.[15] These actions are listed below as found in the specification document. Steps marked in bold indicate changes made to accommodate the Boot Odometer mechanism. The entire TPM_Startup specification can be found in the TPM Specification series of documents. It is assumed that as a result of a hard reboot that all PCR's will be set to known initial values.

2. If stType = TPM_ST_CLEAR

    a. Inspect the contents of PCR[8].

        i. If the Boot Status Indicator is found, disable the TPM

    b. Ensure that sessions associated with resources TPM_RT_CONTEXT, TPM_RT_AUTH and TPM_RT_TRANS are invalidated

    c. Reset PCR values to each correct default value

    d. Set the contents PCR[8] to the Boot Status Indicator

    e. Increment the Boot Odometer Value by 1

    f. Set the following TPM_STCLEAR_FLAGS to their default state

        i. PhysicalPresence

        ii. PhysicalPresenceLock

        iii. disableForceClear

    g. The TPM MAY initialize auditDigest to NULL

i. If not initialized to NULL the TPM SHALL ensure that auditDigest contains a valid value

ii. If initialization fails the TPM SHALL set auditDigest to NULL and SHALL set the internal TPM state so that the TPM returns TPM_FAILED_SELFTEST to all subsequent commands.

h. The TPM SHALL set TPM_STCLEAR_FLAGS -> deactivated to the same state as TPM_PERMANENT_FLAGS-> deactivated

i. The TPM MUST set the TPM_STANY_DATA fields to:

j. TPM_STANY_DATA->contextNonceSession is set to NULLS

ii. TPM_STANY_DATA->contextCount is set to 0

iii. TPM_STANY_DATA->contextList is set to 0

k. The TPM MUST set TPM_STCLEAR_DATA fields to:

i. Invalidate contextNonceKey

ii. countID to NULL

iii. bGlobalLock to FALSE

l. Determine which keys should remain in the TPM

m. For each key that has a valid preserved value in the TPM

(1) if parentPCRStatus is TRUE then call TPM_FlushSpecific(keyHandle)

(2) if IsVolatile is TRUE then call TPM_FlushSpecifid(keyHandle)

Figure 5. Behavior of TPM_Startup when signaled with the TPM_ST_CLEAR flag after [15]

The test for the Boot Status Indicator occurs at step *a* directly before the PCR's are cleared. This is because PCR[8] is used as the container for the Boot Status Indicator. The Boot Status Indicator is an arbitrary binary string that is no larger than a PCR. It is always the same. If the Boot Status Indicator is discovered inside PCR[8], it is assumed

that the computer did not undergo a hard reboot. This assumption is made because if the computer was hard rebooted, power would be cut to the TPM and register contents would be cleared. If a soft reboot is found to have occurred, the TPM is disabled. This disabling does not prevent the host platform from accessing the hardware accelerated encryption algorithms provided by the TPM.[8] It merely prevents the host platform from accessing services that are associated with TPM protected encryption keys. The TPM is not re-enabled until the computer is hard rebooted. If the computer is running in a TPM enabled state, any programs can implicitly assume the computer was started with a hard reboot.

In the case where the Boot Status Indicator is not found, it is assumed that the computer underwent a hard reboot. This is assumed because immediately after power-on, the initial default state of the PCR is a known default value of all 0's.[16] Since the contents of PCR[8] are cleared at step *c*, the Boot Status Indicator is re-entered into PCR[8] at step *d* to ready it for the next reboot event. At step *e* the Boot Odometer Value is incremented to keep track of the number of times the computer has been hard rebooted. No count is kept for the number of soft reboots. Justifications for the Boot Odometer Value functionality can be found in the section labeled Use Cases.

The Boot Odometer Value must be stored in the TPM's long-term (i.e. non-volatile) storage. Because the choice of physical components used to provide non-volatile storage is left to individual TPM implementers, we do not concern ourselves with the requirements for dealing with those components. It is sufficient to note that space must be allocated in the TPM's non-volatile storage for the Boot Odometer Value and that the Value's integrity must be assured by the TPM. The tamper resistance properties of the TPM determine the integrity of the Boot Odometer Value. Tamper resistances requirements are outlined in the TPM specification.[10]

D. SOFTWARE DESIGN

A fully operational TPM is evidence that a host has undergone a hard reboot. This is because a platform that has not undergone a hard reboot will have its TPM disabled. Therefore, any local program wishing to verify that the computer has undergone a hard reboot can query the TPM to ensure that it is enabled. However, the converse is not true. A TPM maybe disabled for several reasons. A program cannot assume that a disabled TPM indicates a soft reboot.

To obtain the current Boot Odometer Value, software on the CPU will need to query the TPM using a new instruction. Because the Boot Odometer functionality does not currently exist in the TPM, a new instruction must be added to the TPM interface. This instruction returns the current Boot Odometer Value, such that it is cryptographically signed by the TPM if requested. In addition, this instruction will be used internally by the TPM during the process of generating attestation responses concerning the current Boot Odometer Value. Since the manufacturers' TPM modules will vary due to implementation differences allowed in the TPM specification, no implementation details of the instruction are included. It is sufficient to describe the new instruction's operation and note that it must exist for the Boot Odometer mechanism to operate completely.Software on the remote server that requests the Boot Odometer Value attestation must be capable of several things. First, it must be able to securely store previously attested Boot Odometer Values. This is a not a concern on high-security, high integrity-systems as secure storage is a defining property of such systems. Less trusted systems equipped with TPM's could utilize the Sealed Storage service provided by a TPM. Second, the remote server must be able to reliably compare the currently attested Boot Odometer Value with the previously attested Boot Odometer Value to ensure that the Boot Odometer Value on the client has increased. Last, the server must be able to validate an attestation response from a client.

E. USE CASES

Besides the object reuse problem discussed in the background chapter, there are several scenarios in which the Boot Odometer Value could be useful, if the mechanism is modified to log soft boots as well as hard. Consider a client PC in corporate network. Policy dictates that computers should only be used for work purposes. No other software should be installed or used. However, it is believed that some users are using bootable CD's to boot into other operating systems to bypass client enforced policy settings. This could be detected by observing that the Boot Odometer Value has been incremented more than expected between concurrent attestations. This would not be conclusive evidence that policy has been violated. The Boot Odometer Value could also be incremented if the system is undergoing reboots because of a malfunction. However, either situation would warrant further investigation.

Further situations arise in which it must be ensured that a host has undergone a reboot regardless of what type. One such scenario is the application of security patches to system software. It is often the case that a system must undergo a reboot after applying relevant security patches. If a remote observer wished to ensure compliance with this procedure, it could request an attestation that a reboot has occurred.

Another scenario is relevant to managers of large data centers. Consider a situation in which a data center loses partial power. Machines may be rebooted, but need some manual intervention to come back to full operational status. A system administrator could request that all remote hosts attest to their current Boot Odometer Values. By comparing the current values to the previous values, the system administrator could determine which machines had lost power and need further attention.

This chapter gave a high level explanation of the how the Boot Odometer mechanism operates. It also detailed the necessary changes to a TPM needed to implement the Boot Odometer mechanism. Software considerations for interacting with new functionality were also discussed. The next chapter discusses the hardware simulations used to demonstrate and test Boot Odometer functionality.

IV. IMPLEMENTATION

A. CODING

The prototype feasibility demonstration for this project was generated using Version 3 of the SimpleScalar software package. This prototype was based on the SimpleScalar/PISA target architecture. This architecture extends the MIPS processor described by Patterson and Hennessy in "Computer Organization and Design"[17] with instructions from the MIPS-IV ISA and RS-6000 instruction set definitions.[18] This simulator does not support privileged processor instructions. Therefore, only user level code can be executed and the prototype is limited in its functionality. The prototype functionality includes a new instruction added to the SimpleScalar/PISA target, a new register, and the simulation of the persistent memory used to store protected information.

The PISA target is assumed to be a generic representation of the TPM module. Although currently lacking the major cryptographic services provided by the TPM, the PISA target is assumed to logically simulate the general-purpose computation carried out within the TPM.

1. Concept of Operation

The prototype implementation operates differently from the conceptual design. This is due to limitations within the SimpleScalar software; it does not currently support full system simulation. It only executes non-privileged instructions. Furthermore, it does not support the concept of a boot up process. This prototype implements the instruction for retrieving the value from the Boot Odometer Value from a PCR. Because a full PCR register bank was not implemented, a single register called the Boot Odometer Register was implemented instead. The Boot Odometer Register is separated from the other registers so that it cannot be addressed by instructions that access general purpose registers. The Boot Odometer load process and the Boot Odometer Value store and increment procedures are implemented as software procedures within the simulator. The simulator does not include functionality for a secure coprocessor such as the TCP TPM.

The simulation operates in the following manner. The simulator begins by retrieving the current Boot Odometer Value from a file that represents the non-volatile

storage of the TPM. It then increments the Boot Odometer Value and writes it back to the file representing the non-volatile storage. Next, it copies the Boot Odometer Value into the Boot Odometer Register. At this point, the simulator starts execution of a binary program. The program is used to test the operation of the *sst* instruction and Boot Odometer Register. The binary program inspects the value present in the Boot Odometer Register and copies it to a general-purpose register using the *sst* instruction. The *sst* instruction was implemented as part of the sim-bor simulator. The program then copies the register value, which is now the Boot Odometer Value, to a variable in memory. The variable is then printed to the standard output for inspection.

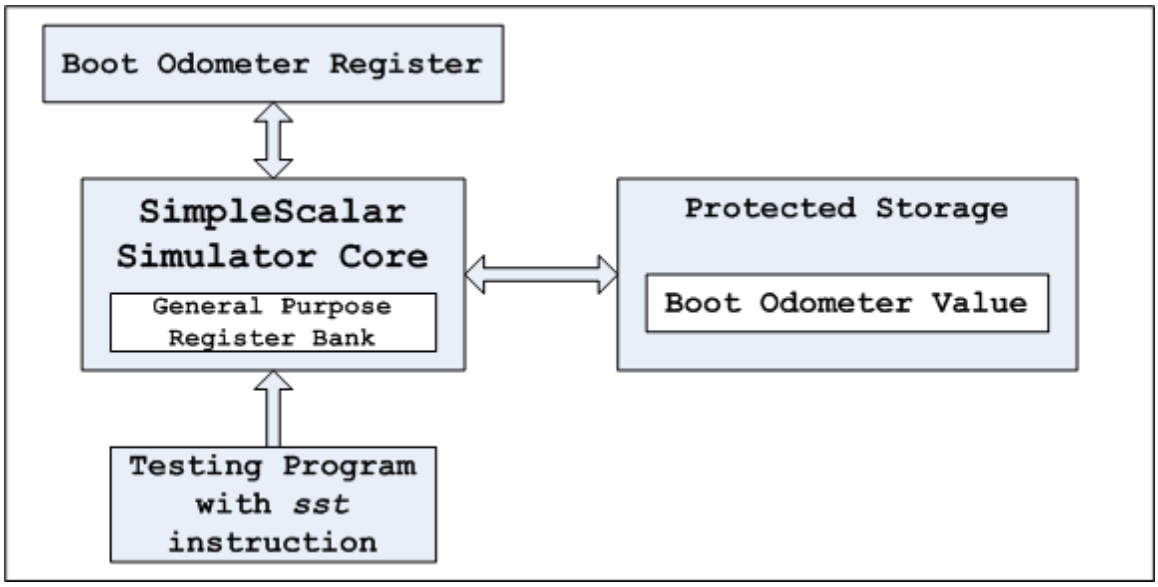

Figure 6. Logical Architecture of Enhancements to SimpleScalar Simulator

2. Instruction Addition

All instructions for a SimpleScalar supported ISA are defined in a DEF file associated with that ISA.[19] The DEF file includes all the semantics of the instruction as well as a functional implementation that describes the behavior of the instruction. These behavioral descriptions are written in C with well-defined macros for accessing processor state. The instruction added for this project has been named "*sst*" for SimpleScalar Test. This instruction copies the value of the Boot Odometer Register to a general-purpose register specified in the instruction. The Boot Odometer Value can then be accessed from the designated general-purpose register. This indirect access is required since the Boot

Odometer Register cannot be addressed by instructions that access general purpose registers.

To access the processor state for the BOR, two new C instruction macros were added to the simulator. These macros read from and write to the processor. The macros implement code that directly access the state variables associated with the simulated TPM.

3. Register Bank

In this implementation, the SimpleScalar simulation architecture was extended to include the new Boot Odometer Register. SimpleScalar utilizes a shared register construction for all ISA implementations currently supported by the version used. The register file includes 32 general-purpose registers, 32 floating-point registers, Hi/Low result registers, a floating-point control register, program counter register and a next program counter register. The register bank is represented in memory with a C structure that contains each of the registers. The simulator stores the Boot Odometer Value as an unsigned integer.

4. Sim-bor

Sim-bor is an extended version of sim-safe that implements functionality to be used in conjunction Boot Odometer Register modeling. Sim-safe is an in-order sequential instruction processor simulator in the SimpleScalar toolset. The sim-bor extension includes a small piece of code that accesses the file that represents the “protected store” and copies the value in the file directly to the variable that represents the Boot Odometer Register. The value is then incremented by one and written back to the file. At this point, the Boot Odometer Register appears to be loaded previous to the any action taken by the program being run on the processor.

5. Protected Storage

A file represents the protected storage. Since a trusted co-processor simulator is not available, this part of the system was not fully included in the SimpleScalar implementation of the Boot Odometer.

6. Cross Compiling & Op Code Insertion

To generate code to run on a SimpleScalar processor emulator, a cross compiler is required. A cross compiler executes on a processor with a given ISA but generates code

for a different processor with a different ISA. The GNU C compiler was configured to act as the cross compiler.

Since there are no C instructions or PISA assembler instructions for the new *sst* instruction, the binary op code to execute the instruction must be crafted by hand and inserted into the execution stream of a program. Instead of writing the entire program in assembler to control register access, the op code was inserted using in-line assembler that can be handled by gcc. This allowed the rest of the program to be easily written in C.

B. TESTING

1. Test Plan

The implementation in SimpleScalar can be tested both interactively and using scripts. For testing this prototype, correct operation is defined in the following manner. After the emulator starts, but before code execution begins within the emulator, the emulator loads the Boot Odometer Register from the file representing the protected store. The value is then incremented by the emulator and written back to the protected store. The software execution begins and the *sst* instruction to access the Boot Odometer Register is called. The Boot Odometer Value is copied to a general-purpose register specified by the calling instruction. The Boot Odometer Value can then be examined arbitrarily. Correct execution is assumed if the value within in the specified general-purpose register has increased by the increment amount in relation to the previously stored value. In this implementation, the increment amount is one.

a. Interactive Testing Procedure

The SimpleScalar toolset includes a debugger named DeLite. This is a simple debugger with basic functionality such as setting break points, code disassembly, and memory inspection. This tool can be used to inspect programs running on the emulated Simple Scalar processors to examine internal processor state. A sample screen shot of DeLite can be seen Figure 7. By examining the state of the processor when testing the Boot Odometer Register, correct execution can be confirmed by ensuring the that the Boot Odometer Register value matches the value read in from the protected storage.

Interactive testing required a test program that used the instruction that accesses Boot Odometer Register to query its current value. The program then displayed the value found in the register.

The first time the processor was inspected took place directly after the SimpleScalar sim-bor simulator loaded the program. At this point, it can be confirmed that the simulator properly read the Boot Odometer Value from the file that acts as the protected store for the simulator. The second time the processor will be inspected is directly before the *sst* instruction is executed. At this point, it can be confirmed that the program has not altered the Boot Odometer Value currently stored. Confirming the integrity of the Boot Odometer Value ensures that the following step occurred correctly. Directly after the *sst* instruction is called, the destination register is checked to ensure the Boot Odometer Value has been transferred correctly. The value inside the destination register specified by the *sst* instruction will be printed to the screen using a function call to *printf()*.

```
sim: ** starting functional simulation **
[          0] 0x00400140:    lw        r16,0(r29)
DLite! > dis 0x400140
    0x00400140:   lw        r16,0(r29)
    0x00400148:   lui       r28,0x1001
    0x00400150:   addiu     r28,r28,-27040
    0x00400158:   addiu     r17,r29,4
    0x00400160:   addiu     r3,r17,4
    0x00400168:   sll       r2,r16,2
    0x00400170:   addu      r3,r3,r2
    0x00400178:   addu      r18,r0,r3
    0x00400180:   sw        r18,-32428(r28)
    0x00400188:   addiu     r29,r29,-24
    0x00400190:   addu      r4,r0,r16
    0x00400198:   addu      r5,r0,r17
    0x004001a0:   addu      r6,r0,r18
    0x004001a8:   jal       0x400500
    0x004001b0:   sw        r0,-32432(r28)
    0x004001b8:   addu      r4,r0,r16
DLite! >
```

Figure 7. Example of partial source disassembly using the SimpleScalar DLite! debugger.

b. Scripted Testing Procedure

Because the interactive testing can only reasonably inspect a limited range of values during operation, scripted testing was included. However, full value ranges

cannot be tested exhaustively using automated test procedures. Therefore, the range 4,294,767,295 - 4,294,967,295 was tested. These values represent the 200,000 integer values directly preceding the overflow value of a 32-bit unsigned integer. In addition, the overflow condition was tested immediately following the testing of this block. The expected behavior is that register should roll over to zero and continue functioning without interruption.

Automated testing was done using a Python script that repeatedly executes the Boot Odometer simulator. The simulator ran a small program that executes the *sst* instruction implemented within the simulator. The script parsed the output from both the simulator and the program being run by the simulator. The script compared two values. The first is the value of the simulated TPM non-volatile storage. This was obtained as debugging information provided by the simulator. The second was the value returned by the execution of the *sst* instruction. The script tested the two values to ensure that they were equal. If they were not, the script would exit. This check ensures that the *sst* instruction correctly retrieved the correct value from the Boot Odometer Register. The script also checks to ensure that the Boot Odometer Value is exactly one higher than the value in the previous test cycle. This ensures that the Boot Odometer only increases by a value of one.

2. Test Results

a. Interactive Test Results

The interactive testing did not show any unpredicted behavior.

b. Scripted Test Results

The scripted test results did not show any unpredicted behavior.

This chapter discussed the operation of the simulation used to demonstrate and test the operation of the Boot Odometer mechanism. The SimpleScalar simulator suite was modified to simulate the functionality of the Boot Odometer mechanism. The simulation software was shown to be flexible in its ability handle changes incorporated for the this thesis. Further, the tests showed that the changes operated as intended.

The next chapter discusses security issues concerning the implementation of the Boot Odometer mechanism. The topics covered are covert channel analysis, instruction privilege, and denial of service.

THIS PAGE INTENTIONALLY LEFT BLANK

V. SECURITY ANALYSIS

A. COVERT CHANNEL ANALYSIS

It is not possible to exploit the Boot Odometer Value as a useful covert channel. Because the Boot Odometer Value only increments in cases where the computer is hard rebooted and there is no way for a software program to initiate a hard reboot, a malicious program cannot influence the Boot Odometer Value for use as a covert channel.

While a *person* could attempt to create a covert channel by manually selecting a software or hardware reboot, he could only signal to an observing program a single bit of sensitive information at a time. Assuming a relatively fast boot process of 15 seconds, a user could signal approximately four bits a minute. However, since the user already has access to the sensitive information, he does not need the computer (and covert channel) to help him compromise the information (this is why cover channel analysis focuses on the actions of programs rather than users.).

B. INSTRUCTION PRIVILEGE

The section titled “Software”, in Chapter III, discussed the need for an additional instruction to be added to the TPM that would return the current Boot Odometer Value. It is reasoned here that there is no need for this instruction to be constrained only to privileged processes. The first reason is that there are no logically feasible covert channels to be exploited in conjunction with the Boot Odometer Value. Second, the TPM protects the integrity of the Boot Odometer Value. There is no way for untrusted processes to affect the Boot Odometer Value through the use of the instruction. Therefore, the instruction used to retrieve the Boot Odometer Value from the TPM does not need to be privileged.

C. DENIAL OF SERVICE

Because the function call for accessing of the Boot Odometer Value is accessible to unprivileged users, it is possible that a malicious user may attempt to perform a denial of service attack on the TPM by flooding it with requests to perform the function. This is no different than any other non-privileged TPM call. The Trusted Software Stack (TSS)

mitigates a denial of service attack. The TSS is an API that allows applications a high level interface to the TPM functionality and prevents raw access to the TPM by untrusted applications.[9] By acting as a gatekeeper to the TPM, the TSS prevents it from being overwhelmed.

VI. CONCLUSION

The ability to sense the reboot type by modifying the Trusted Platform Module is feasible as demonstrated by the explanation of both the conceptual design and the hardware simulations. The TPM provides trusted remote attestation that allows the MYSEA client to prove to the MYSEA Server that it has undergone a hard reboot. This ensures the MYSEA Server that sensitive information is not leaked to lower level process through volatile memory, such as main system RAM, when a MYSEA client switches to a lower or non-comparable classification level.

Through analysis, the conceptual design has been shown to be correct because it is understandable, cannot be bypassed, and cannot by modified. The design can be considered understandable because of its simplicity. It included only four new requirements to the TPM_Startup function. The behavior of each change as well as their combined behavior is easily understood. The new operations cannot be bypassed because they are part of TPM function TPM_Startup. This function occurs during every system boot cycle and is specified in TPM design documents. The added functionality is non-modifiable because it is protected by the TPM.

The simulation of the TPM functionality as related to the Boot Odometer concept shows that the operation of such a concept is feasible. The simulation allowed for a low cost, flexible, and testable design. Further, we were able to show that the Boot Odometer concept had uses outside of the MYSEA project and could be easily used in any situation where a reboot needs to be confirmed.

Given further resources, several avenues of research can be used to extend the work done here. First, a full specification of a commercially produced TPM would need to be obtained. Using that specification, a feasibility study on the integration of the Boot Odometer concept could be completed. If a developer's model of a TPM was available and its firmware was upgradeable, the changes could be integrated into a working TPM and tested. Second, using commercially available TPM-enabled computers, work on integrating attestation into the MYSEA framework could be done. While the Boot

Odometer mechanism would not be available, other attestation scenarios in which attestation would be beneficial could be tested.

In conclusion, the Boot Odometer concept is a useful design with functionality relevant to the MYSEA project. Furthermore, it is shown the conceptual design and simulation testing demonstrates the feasibility of an implementation of this concept.

APPENDIX A

A. SIMPLESCALAR SETUP ON FEDORA CORE 2[19]

1. Download the following tarballs at the SimpleScalar website - simpleutils-990811.tar.gz, simplesim-3v0d.tgz, gcc-2.7.2.3.ss.tar.gz, and simpletools-2v0.tgz. (The direct URL is http://simplescalar.com/tools.html.)
2. Create a directory for installation referred to here as $IDIR. Move the downloaded files into $IDIR and extract each tarball.
3. Execute the following commands for installation of the Build Utils.
   export IDIR=/(chosen installation directory)
   cd $IDIR/simpleutils-990811
   ./configure --host=i386-*-linux --target=sslittle-na-sstrix --with-gnu-as --with-gnu-ld --prefix=$IDIR
4. Execute the following commands to install SimpleScalar v3.0
   cd $IDIR/simplesim-3.0
   make config-pisa
   make
   make sim-tests
5. Due to unknown problems in the source distributions of this version of *gcc*, several of the files must be modified for compilation to complete successfully.
   cd $IDIR/gcc-2.7.2.3
   export PATH=$PATH:$IDIR/sslittle-na-sstrix/bin
   ./configure --host=i386-*-linux --target=sslittle-na-sstrix --with-gnu-as --with-gnu-ld --prefix=$IDIR --enable-languages=c,c++
   make
6. At some point, the *make* will fail due to an error and stop compilation. The followings changes should cause the *make* to complete successfully.
   1. Open insn-output.c in a text editor. Ensure that all multi-line quotes that are broken over several lines are properly escaped using the correct escape character. In this case – "\".
   2. Replace the $IDIR/sslittle-na-sstrix/include/sys/cdefs.h file with the cdefs.h file found in the directory named "patched"
   3. On line 35 of obj/sendmsg.c, add the following –
      "#define STRUCT_VALUE 0"
   4. One line 60 of protoize.c, replace the "#include <varargs.h>" with "#include <stdarg.h>."
7. At this point, the compilation should be able to continue successfully.
   make
   make enquire
   ../simplesim-3.0/sime-safe ./enquire –f > floah.h-cross
   make install
8. Building glibs is not necessary. A working version exists in $IDIR/sslittle-na-sstrix/lib/libc.a

B. SIMPLESCALAR DEVELOPMENT ENVIRONMENT

1. Binary Utilities

When working with SimpleScalar, it is highly probable that the host system architecture is different than that of the processor architecture being simulated by SimpleScalar. The ISA of the emulated processor may not exist physically or a system with the processor type may not be available to the researcher. In this case, the host architecture must be relied on to host the utilities to work with programs that are complied to run on the SimpleScalar emulated processors. SimpleScalar deals with this problem by including the source for basic utilities needed for working with binary files such as objdump, strings, and strip. These utilities have been ported to work on binaries that are not compiled for the host system. After following the installation instructions, these utilities can be found in the $IDIR/bin directory.

2. Cross Compiling

Instructions for building *gcc* as a cross compiler were included in the installation directions. However, since binaries cannot link against host system libraries for C library function calls, the needed code from the libraries is statically compiled into the binary. For unknown reasons, the compiler links in all functions defined by the included header file and not just the functions utilized by the compiled binary. The cross compiler version of gcc is found in the $IDIR/bin directory.

Cross compiling requires that several other hurdles be overcome in the SimpleScalar environment. First, instructions may need to be used in target architecture binaries that are not supported by either the compiler or the assembler. Second, attempting to hand craft op codes and insert them into compiler-generated assembly code may overwrite register contents. Thus, there needs to be a way for the compiler to choose the registers used by any new instructions used in binaries. Both of these problems can be addressed with the use of inline assembly code. Inline assembly code is supported by the *gcc* compiler and allows assembly code to be placed inline with C code in a source file. Furthermore, this allows the compiler to choose the registers needed for binary optimization. Since the assembler won't support any new instructions, hand crafted op codes are used to include them. Two methods are suggested by SimpleScalar documentation.[20] The first is illustrated below.

```
int multiply_int32(int a, int b)
{
     int c;

     asm(“.long ((0x01 << 25) | (0x00 << 23) |
     ((XXX%0) << 9) | ((XXX%1) << 4) | (XXX%2))” :
     “=r” (c) : “r” (a) , “r” (b));

     return c;
}[20]
```

This method displays the inline assembler imbedded in a function call. Another form is to embed the op code in a macro function.

```
#define __multiply_int32(a,b)({ int c; asm(“.long
((0x01 << 25) | (0x00 << 23) | ((XXX%0) << 9) |
((XXX%1) << 4) | (XXX%2))” : “=r” (c) : “r” (a) , “r”
(b)); c;})[20]
```

Several more steps are required to finish cross compiling a binary that includes inline assembler following these examples.

```
sslittle-na-sstrix-gcc –O –S main.c
sed “s/XXX//g” main.s > main_fix.s
sslittle-na-sstrix-ggc –O –c main_fix.s
```

If wanted, the sslittle-na-sstrix-objdump utility can be used to examine the binary for the correct inclusion of the op code.

3. Simulation

The SimpleScalar toolset contains a suite of processor simulators. This project utilizes a variant of the sim-safe simulator, called sim-bor, that has been adapted for use in the Boot Odometer prototype. Sim-bor is a functional simulator that uses serial execution of instructions. Sim-bor runs from the command line. It takes several arguments. The final argument is the name of binary to be executed using the simulator. Included with the name of the binary are any arguments it takes. When execution of the binary has been completed, sim-bor outputs statistics related to the execution of the binary.

4. Debugger

Running sim-bor with the –i argument causes the target binary to be run in the DLite! debugger. The DLite! debugger allows the programmer to step through the target binary by setting and running breakpoints. Also, the debugger supports commands that

access value of memory address and registers. It also allows for access to simulator statistics during execution.

APPENDIX B

A. SIMPLESCALAR DIFFS

The standard Unix *diff* utility was used to create a listing if the changes made to the SimpleScalar source code when implementing this thesis. The output of the *diff* utility is listed before. The output can be combined with a fresh install of the SimpleScalar source to reconstruct the changes made for this thesis.

```
Only in simplesim-3.1: boot_mem
Only in simplesim-3.1: dlite.o
Only in simplesim-3.1: eio.o
Only in simplesim-3.1: endian.o
Only in simplesim-3.1: eval.o
Only in simplesim-3.1/libexo: exolex.o
Only in simplesim-3.1/libexo: libexo.a
Only in simplesim-3.1/libexo: libexo.o
Only in simplesim-3.1: loader.o
diff -r simplesim-3.0/machine.c simplesim-3.1/machine.c
105a106
>   "fu-BO-boot",
282a284,286
>
>   /* Boot Odometer */
>   { "$bo",   rt_bor,        0 },
285a290
>
421a427,438
>
>  /* New Stuff added by me*/
>     case rt_bor:
>       if(!is_write)
>         {
>    val->type = et_uint;
>    val->value.as_uint = regs->regs_B;
>    }
>       else
>         regs->regs_B = eval_as_uint(*val);
>       break;
> /* ----------------------*/
diff -r simplesim-3.0/machine.def simplesim-3.1/machine.def
1225a1226,1260
> /*
>  * Boot Odometer DEFINST and Implementation
```

```
>   * The "sst" instruction writes the value in the boot
odometer register
>  * to the general purpose register defined in rd register
field.
>   * The macros BOR and SET_BOR are specially defined for
this project.
>   * They must be defined in each individual simulator in
which they are
>  * to be used.  This project only makes use of the sim-
safe simulator.
>  */
>                      #define                         SST_IMPL
\
>                                                          {
\
>     SET_GPR(RD,BOR);                                      \
>     printf("BOR...%u\n", BOR);
     \
>   }
> /*
>   * 1. This is the name of the instruction.  It is used
for instruction
>  *    decoding in the machine.def file.
>  * 2. 0xbb is the hex value of the opcode.
>  * 3. "sst" is the name of the opcode. this field is used
by the debugger
>   * 4. This section describes the rest of the fields in
the instruction
>  *    to the debugger.  Here we see that the instruction
is described as
>  *    using the rd output register field.
>   * 5. BootOD is a machine resource descriptor on which
the instruction
>  *    depends.  It has been added for this project.
>    * 6. This section describes the flags set by the
instruction.  F_CTRL is
>  *    the control register.
>    * 7. These are the output register dependencies for
advanced simulators.
>    * 8. These are the input register dependencies for
advanced simulators.
>  */
> DEFINST(SST,/*1*/      0xbb,/*2*/
>         "sst",/*3*/            "d",/*4*/
>         BootOD,/*5*/            F_CTRL,/*6*/
>         DGPR(RD), DNA,/*7*/     DNA, DNA, DNA)/*8*/
>
>
```

```
>
2186a2222
> #undef SST_IMPL
Only in simplesim-3.1: machine.def.new
Only in simplesim-3.1: machine.def.old
Only in simplesim-3.1: machine.def.old.2
diff -r simplesim-3.0/machine.h simplesim-3.1/machine.h
140a141,143
> /* New Stuff ... */
> #define MD_NUM_BREGS        1
>
143c146,149
<   (/*int*/32 + /*fp*/32 + /*misc*/3 + /*tmp*/1 + /*mem*/1
+ /*ctrl*/1)
---
>   (/*int*/32 + /*fp*/32 + /*misc*/3 + /*tmp*/1 + /*mem*/1
+ /*ctrl*/1 + /*boot*/1)
>
> /* boot odometer register file entry type */
> typedef sword_t md_bor_t;
265a272
>   BootOD,         /* Boot Odometer */
596c603,604
<   rt_NUM
---
>   rt_NUM,
>   rt_bor          /* boot odometer register */
Only in simplesim-3.1: machine.o
Only in simplesim-3.1: main.o
diff -r simplesim-3.0/Makefile simplesim-3.1/Makefile
374a375,377
> sim-bo$(EEXT):    sysprobe$(EEXT)  sim-bo.$(OEXT)  $(OBJS)
libexo/libexo.$(LEXT)
>    $(CC)   -o   sim-bo$(EEXT)   $(CFLAGS)   sim-bo.$(OEXT)
$(OBJS) libexo/libexo.$(LEXT) $(MLIBS)
>
476a480,481
>  sim-bo.$(OEXT):  host.h  misc.h  machine.h  machine.def
regs.h memory.h
>  sim-bo.$(OEXT):  options.h  stats.h  eval.h  loader.h
syscall.h dlite.h sim.h
Only in simplesim-3.1: Makefile.old
Only in simplesim-3.1: memory.o
Only in simplesim-3.1: misc.o
Only in simplesim-3.1: options.o
Only in simplesim-3.1: range.o
diff -r simplesim-3.0/regs.c simplesim-3.1/regs.c
161a162,164
```

```
> /* boot odometer */
> SS_WORD_TYPE regs_B;
>
diff -r simplesim-3.0/regs.h simplesim-3.1/regs.h
105a106
>   md_bor_t regs_B;          /*   boot   odometer   register
file */
Only in simplesim-3.1: .regs.h.swp
Only in simplesim-3.1: regs.o
Only in simplesim-3.1: sim-bo
Only in simplesim-3.1: sim-bo.c
Only in simplesim-3.1: sim-bo.o
Only in simplesim-3.1: sim-fast
diff -r simplesim-3.0/sim-fast.c simplesim-3.1/sim-fast.c
253a254,255
> #define BOR            (regs.regs_B)
> #define SET_BOR(EXPR)       (regs.regs_B = (EXPR))
Only in simplesim-3.1: sim-fast.o
Only in simplesim-3.1: sim-safe
diff -r simplesim-3.0/sim-safe.c simplesim-3.1/sim-safe.c
216a217,218
> #define BOR            (regs.regs_B)
> #define SET_BOR(EXPR)       (regs.regs_B = (EXPR))
Only in simplesim-3.1: sim-safe.o
Only in simplesim-3.1: stats.o
Only in simplesim-3.1: symbol.o
Only in simplesim-3.1: syscall.o
Only in simplesim-3.1: sysprobe
diff    -r    simplesim-3.0/target-pisa/pisa.c    simplesim-
3.1/target-pisa/pisa.c
105a106
>   "fu-BO-boot",
282a284,286
>
>   /* Boot Odometer */
>   { "$bo",   rt_bor,        0 },
285a290
>
421a427,438
>
>  /* New Stuff added by me*/
>     case rt_bor:
>       if(!is_write)
>         {
>    val->type = et_uint;
>    val->value.as_uint = regs->regs_B;
>    }
>       else
```

```
>         regs->regs_B = eval_as_uint(*val);
>       break;
> /* ----------------------*/
Only in simplesim-3.1/target-pisa: pisa.c.old
Only in simplesim-3.1/target-pisa: .pisa.c.swp
diff   -r   simplesim-3.0/target-pisa/pisa.def   simplesim-
3.1/target-pisa/pisa.def
1225a1226,1260
> /*
>  * Boot Odometer DEFINST and Implementation
>   * The "sst" instruction writes the value in the boot
odometer register
>  * to the general purpose register defined in rd register
field.
>   * The macros BOR and SET_BOR are specially defined for
this project.
>   * They must be defined in each individual simulator in
which they are
>  * to be used.  This project only makes use of the sim-
safe simulator.
>  */
>                      #define                         SST_IMPL
\
>                                                          {
\
>     SET_GPR(RD,BOR);                                    \
>     printf("BOR...%u\n", BOR);
     \
>   }
> /*
>   * 1. This is the name of the instruction.  It is used
for instruction
>  *    decoding in the machine.def file.
>  * 2. 0xbb is the hex value of the opcode.
>  * 3. "sst" is the name of the opcode. this field is used
by the debugger
>   * 4. This section describes the rest of the fields in
the instruction
>  *    to the debugger.  Here we see that the instruction
is described as
>  *    using the rd output register field.
>   * 5. BootOD is a machine resource descriptor on which
the instruction
>  *    depends.  It has been added for this project.
>    *  6.  This  section  describes  the  flags  set  by  the
instruction.  F_CTRL is
>  *    the control register.
```

```
>   * 7. These are the output register dependencies for
advanced simulators.
>   * 8. These are the input register dependencies for
advanced simulators.
>  */
> DEFINST(SST,/*1*/      0xbb,/*2*/
>         "sst",/*3*/            "d",/*4*/
>         BootOD,/*5*/            F_CTRL,/*6*/
>         DGPR(RD), DNA,/*7*/     DNA, DNA, DNA)/*8*/
>
>
>
2186a2222
> #undef SST_IMPL
diff    -r    simplesim-3.0/target-pisa/pisa.h    simplesim-
3.1/target-pisa/pisa.h
140a141,143
> /* New Stuff ... */
> #define MD_NUM_BREGS        1
>
143c146,149
<   (/*int*/32 + /*fp*/32 + /*misc*/3 + /*tmp*/1 + /*mem*/1
+ /*ctrl*/1)
---
>   (/*int*/32 + /*fp*/32 + /*misc*/3 + /*tmp*/1 + /*mem*/1
+ /*ctrl*/1 + /*boot*/1)
>
> /* boot odometer register file entry type */
> typedef sword_t md_bor_t;
265a272
>   BootOD,         /* Boot Odometer */
596c603,604
<   rt_NUM
---
>   rt_NUM,
>   rt_bor          /* boot odometer register */
```

B. SIMPLESCALAR TEST SCRIPT

This script was used for automated testing of SimpleScalar changes made in support of this thesis.

```
#The sst instruction copies the value of the boot odometer
register to
#a general purpose register. From here, the Boot Odometer
Value can
#be inspected by no privileged processes.
```

```
import os
import re

#Since the Boot Odometer Value is a 32-bit unsigned int,
every value
#cannot be tested. We will test a 200000 value block before
the integer
#rollover point. The rollover point will also be done to
test the
#boundry behavior.

high = pow(2,32)
low = high - 200000

l = low - 2
j = low - 2

for i in range(low, high):

   if (i%1000) == 0:
      print "."

   #Store the values from the previous loop iterations.
Compare them
   #to the new values to make sure the value has been
incremented by
   #one
   l_prev = l
   j_prev = j

   #This runs the simulator and stores the output in the a
object
   a    =    os.popen3("/home/nexus/SimpleScalar/simplesim-
3.1/sim-bo /home/nexus/SimpleScalar/test/test2", "r")

   #The output we want comes on stdout. It's been specially
added
   #the simulator code.
   out = a[1]

   #SimpleScalar simulators output their debugging messages
to stderr
   #We're not interested in that info so it will be
ignored.
   #err = a[2]

   #Extract the string data from the programs stdout
```

```
   z = out.read()

   #Generate the regular expressions matching patterns
   p = re.compile('BOR...([0-9]*)')
   q = re.compile('sst...([0-9]*)')

   #Search the strings for the regex patterns
   m = p.search(z)
   n = q.search(z)

   #Extract the Boot Odometer values that we're interested
in.
   val_bor = m.group(1)
   val_reg = n.group(1)

   #Convert the string representations of the observed
register values
   #into integers
   l = int(val_bor)
   j = int(val_reg)

   #make sure the value read directly from the register
memory file
   #and the value obtained by executing the sst instruction
are the
   #same. If they are not the same, print the values and
exit.
   if l != j:
      print "Value mismatch detected!!!\n"
      os.exit(0)

   #Ensure the the Boot Odometer Value has been
monotomically
   #incremented for both values
   if (l_prev + 1) != l:
      print "Value did not increment correctly...", l_prev,
l, "\n"
      os.exit()
   if (j_prev + 1) != j:
      print "Value did not increment correctly\n"
      os.exit()
```

LIST OF REFERENCES

[1] Nguyen, T. D., Levin, T. E., Irvine, C. E., "MYSEA Testbed," Proceedings from the 6th IEEE Systems, Man and Cybernetics Information Assurance Workshop, West Point, NY, June 2005, pp. 438-439.

[2] TCG Specification Architecture Overview, Rev. 1.2. Available: https://www.trustedcomputinggroup.org/downloads/TCG_1_0_Architecture_Overview.pdf. Accessed: August 2005

[3] Anderson, J. P., Computer Security Technology Planning Study. Technical Report ESDTR-73-51, Air Force Electronic Systems Division, Hanscom AFB, Bedford, MA, 1972.

[4] Irvine, C. E., Shifflett, D. J., Clark, P. C., Levin, T. E., and Dinolt, G. W., "MYSEA Technology Demonstration", DARPA DISCEX Conference, April 2003.

[5] Irvine, C. E., Levin, T. E., Nguyen, T. D., Shifflett, D. J., Khosalim, J., Clark, P. C., Wong, A., Afinidad, F., Bibighaus, D., and Sears, J., "Overview of a High Assurance Architecture for Distributed Multilevel Security", Proceedings of the 2004 IEEE Systems, Man and Cybernetics Information Assurance Workshop, West Point, NY, June 2004.

[6] A Guide to Understanding Object Reuse in Trusted Systems, Report No. NCSC TG-018, National Computer Security Center, Ft. George G. Meade, MD, 1 July 1991.

[7] Schecter, S.E., Greenstadt, R.A., and Smith, M.D., “Trusted Computing, Peer-to-Peer Distribution, and the Economics of Pirated Entertainment,” presented at the 2nd Annual Workshop on Economics and Information Security, College Park, MD, May, 2003.

[8] TPM Main Part 1 Design Principles, Specification Version 1.2, Rev. 85. Available:https://www.trustedcomputinggroup.org/downloads/specifications/mainP1DP_rev85.zip. Accessed: August 2005

[9] Barret, M.F., “Towards an Open Trusted Computing Framework,” Master’s thesis, University of Auckland, Auckland, New Zealand, 2005.

[10] TCG PC Specific Implementation Specification, Version 1.1. Available: https://www.trustedcomputinggroup.org/downloads/TCG_PCSpecificSpecification_v1_1.pdf. Accessed: August 2005

[11] Brickell, E., Camenisch, J., and Chen, L., “Direct Anonymous Attestation,” Technical Report HPL-2004-93, Hewlett-Packard Company, Bristol, June 2004.

[12] Austin, T., Larson, E., and Ernst, D., “SimpleScalar: An infrastructure for computer system modeling.” *IEEE Computer*, vol. 35, no. 2, pp. 59--67, Feb. 2002.

[13] Burger, D.C. and Austin, D.C., "The SimpleScalar tool set, version 2.0," Technical Report CS-TR-97-1342, University of Wisconsin, Madison, June 1997.

[14] Nguyen, T. D., Levin, T. E., and Irvine, C. E., "TCX Project: High Assurance for Secure Embedded Systems", 11th IEEE Real-Time and Embedded Technology and Applications Symposium Work-In-Progress Session, San Francisco, CA, March 2005.

[15] TPM Main Part 3 Commands, Specification Version 1.2, Level 2 Rev. 85. Available:https://www.trustedcomputinggroup.org/downloads/specifications/mainP1DP_rev85.zip. Accessed: August 2005

[16] TCG PC Client Specific TPM Interface Specification (TIS) Version 1.2. Available:https://www.trustedcomputinggroup.org/downloads//downloads/specifications/pcclient/TCG_PCClientTPMSpecification_120_100_FINAL.pdf. Accessed: August 2005

[17] J.L. Hennessy, D.A. Patterson, *Computer Organization and Design: The Hardware/Software Interface*, 2nd edition, Morgan Kaufmann, 1997.

[18] Austin, T., “SimpleScalar 3.0 Release [Announcement].” Available: http://www.simplescalar.com/docs/ANNOUNCE-3.0d.txt

[19] Pan, Y., “SimpleScalar Installation Guide”. Available: http://www.comp.nus.edu.sg/~panyu/simplesim.htm. Accessed: September 2005

[20] Austin, T. “SimpleScalar DEF File Format Overview.” December 2003. Available: http://www.simplescalar.com/docs/README-def.txt Accessed: September 2005

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, VA

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, CA

3. Hugo A. Badillo
   NSA
   Fort Meade, MD

4. George Bieber
   OSD
   Washington, DC

5. RADM Joseph Burns
   Fort George Meade, MD

6. John Campbell
   National Security Agency
   Fort Meade, MD

7. Deborah Cooper
   DC Associates, LLC
   Roslyn, VA

8. CDR Daniel L. Currie
   PMW 161
   San Diego, CA

9. Louise Davidson
   National Geospatial Agency
   Bethesda, MD

10. Vincent J. DiMaria
    National Security Agency
    Fort Meade, MD

11. LCDR James Downey
    NAVSEA
    Washington, DC

12. Dr. Diana Gant
National Science Foundation

13. Jennifer Guild
SPAWAR
Charleston, SC

14. Richard Hale
DISA
Falls Church, VA

15. LCDR Scott D. Heller
SPAWAR
San Diego, CA

16. Wiley Jones
OSD
Washington, DC

17. Russell Jones
N641
Arlington, VA

18. David Ladd
Microsoft Corporation
Redmond, WA

19. Dr. Carl Landwehr
National Science Foundation
Arlington, VA

20. Steve LaFountain
NSA
Fort Meade, MD

21. Dr. Greg Larson
IDA
Alexandria, VA

22. Penny Lehtola
NSA
Fort Meade, MD

23. Ernest Lucier
Federal Aviation Administration
Washington, DC

24. CAPT Deborah McGhee
Headquarters U.S. Navy
Arlington, VA

25. Dr. Vic Maconachy
NSA
Fort Meade, MD

26. Doug Maughan
Department of Homeland Security
Washington, DC

27. Dr. John Monastra
Aerospace Corporation
Chantilly, VA

28. John Mildner
SPAWAR
Charleston, SC

29. Jim Roberts
Central Intelligence Agency
Reston, VA

30. Charles Sherupski
Sherassoc
Round Hill, VA

31. Dr. Ralph Wachter
ONR
Arlington, VA

32. David Wirth
N641
Arlington, VA

33. Daniel Wolf
NSA
Fort Meade, MD

34. Jim Yerovi
NRO
Chantilly, VA

35. CAPT Robert Zellmann

CNO Staff N614
Arlington, VA

36. Dr. Cynthia E. Irvine
Naval Postgraduate School
Monterey, CA

37. Thuy D. Nguyen
Naval Postgraduate School
Monterey, CA

38. Paul C. Clark
Naval Postgraduate School
Monterey, CA

39. Timothy E. Levin
Naval Postgraduate School
Monterey, CA

40. Richard C. Vernon
Civilian, Naval Postgraduate School
Monterey, CA